The Basics of UNIX/Linux

13-2. Linked list

Instructor: Joonho Kwon
jhkwon@pusan.ac.kr
Data Science Lab @ PNU

datalab
data science laboratory

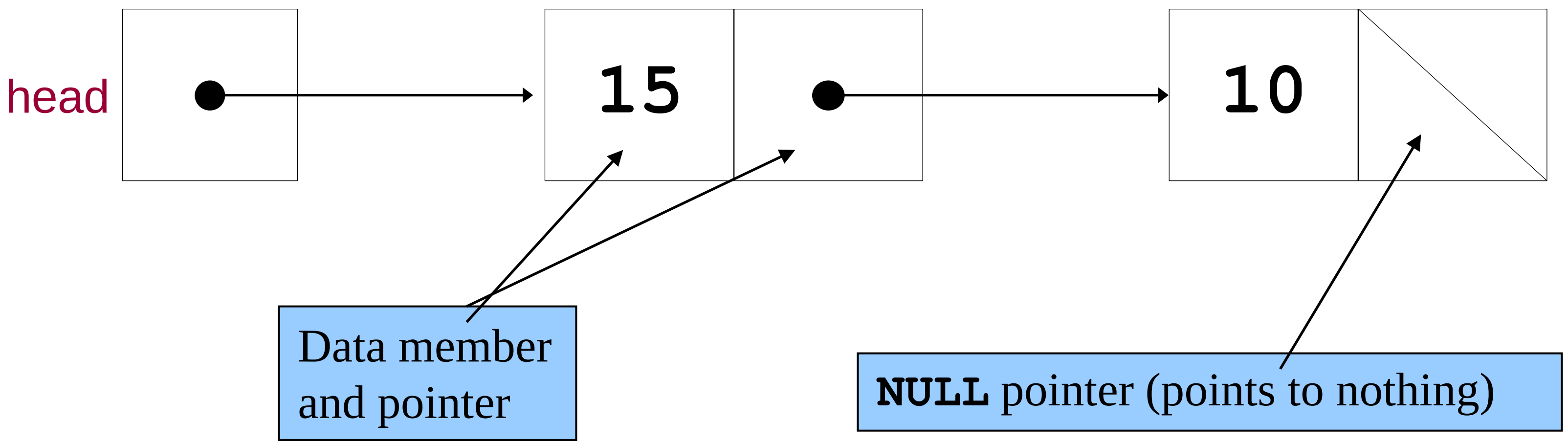
This file is based on slides from Justin Hsia.

Contents

- Self referenced structure
- Implementing Data Structures in C

Self-referential structures (1/2)

- Self-referential structures
  - Structure that contains a pointer to a structure of the same type
  - Can be linked together to form useful data structures such as lists, queues, stacks and trees
  - Terminated with a NULL pointer (0)
- Two self-referential structure objects linked together



Self-referential structures (2/2)

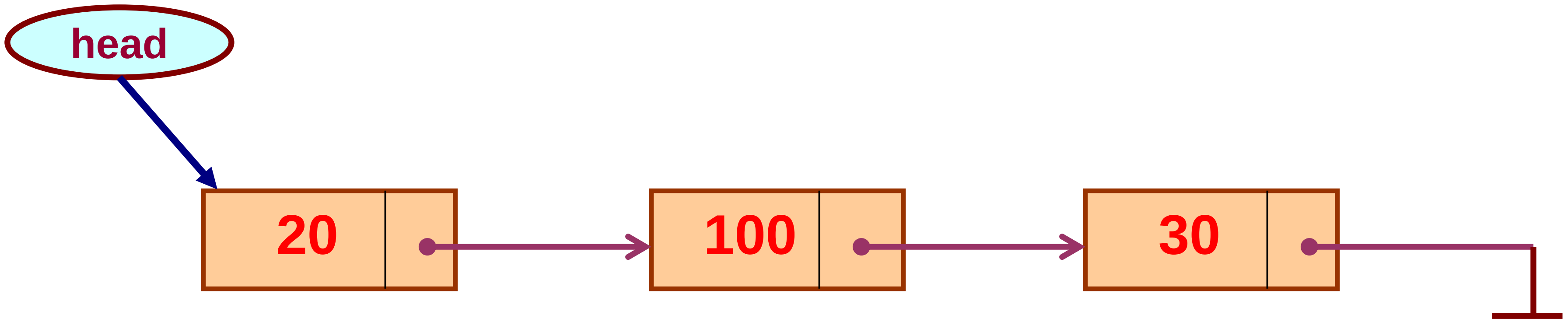
```
struct ListNode {
    int element;
    struct ListNode *next;
}

typedef struct ListNode Node;
```

```
typedef struct ListNode
{
    int element;
    struct ListNode* next;
} Node;
```

- *next - points to an object of type ListNode
  - Referred to as a link – ties one ListNode to another ListNode

Linked List (1/3)

- A linked list is a data structure which can change during execution.
  - Successive elements are connected by pointers.
  - Last element points to `NULL`.
  - It can grow or shrink in size during execution of a program.
  - It can be made just as long as required.
  - It does not waste memory space.



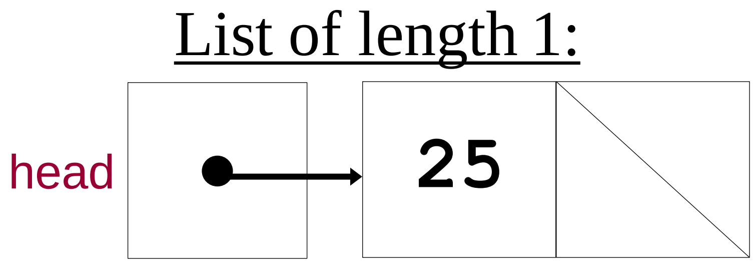
Linked List (2/3)

- Keeping track of a linked list:
  - Must know the pointer to the first element of the list (called *start*, *head*, etc.).

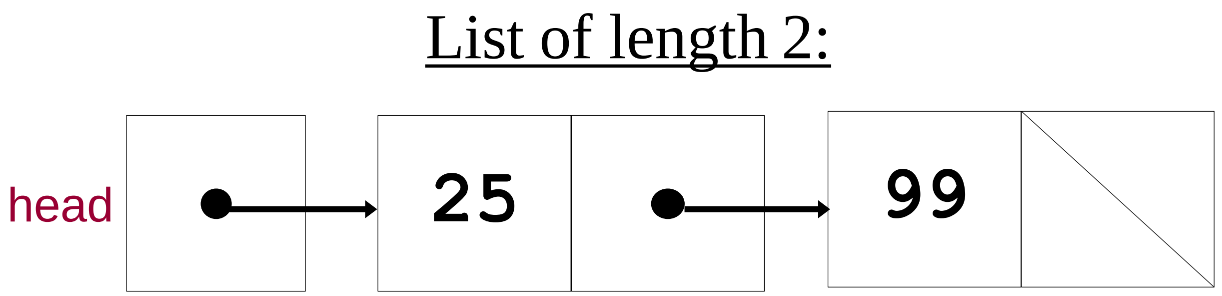
- Linked lists provide flexibility in allowing the items to be rearranged eff iciently.
  - Insert an element.
  - Delete an element.

Linked List (3/3)

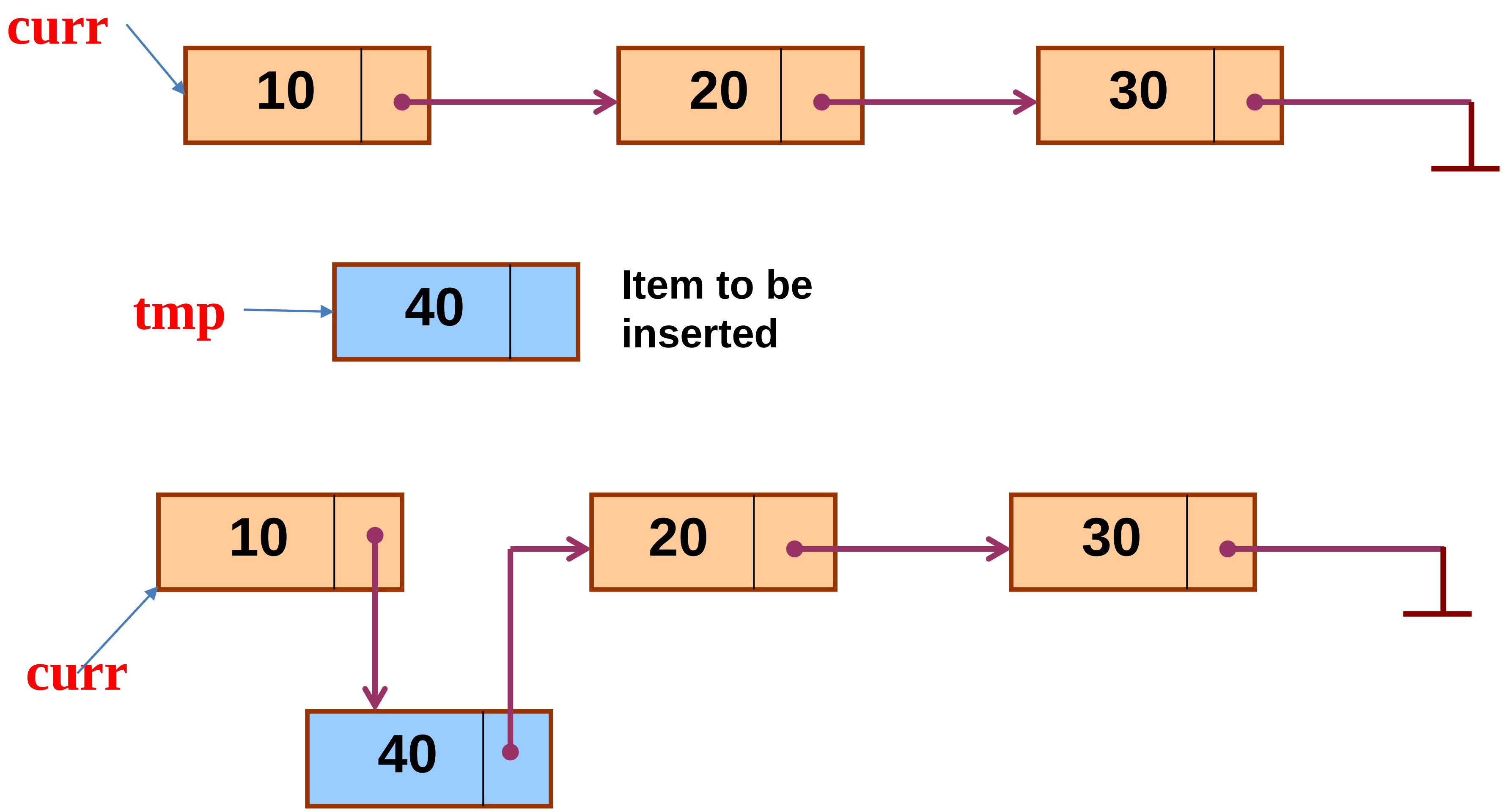
```
node *head;
head = malloc(sizeof(node));
head->element = 25;
head->next = NULL;
```



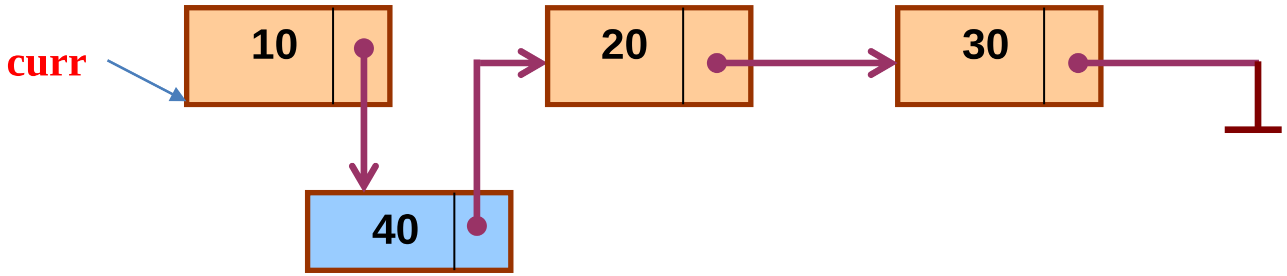

```
head->next = malloc(sizeof(node));
head->next->element = 99;
head->next->next = NULL;
```



Insert in the middle: Illustration

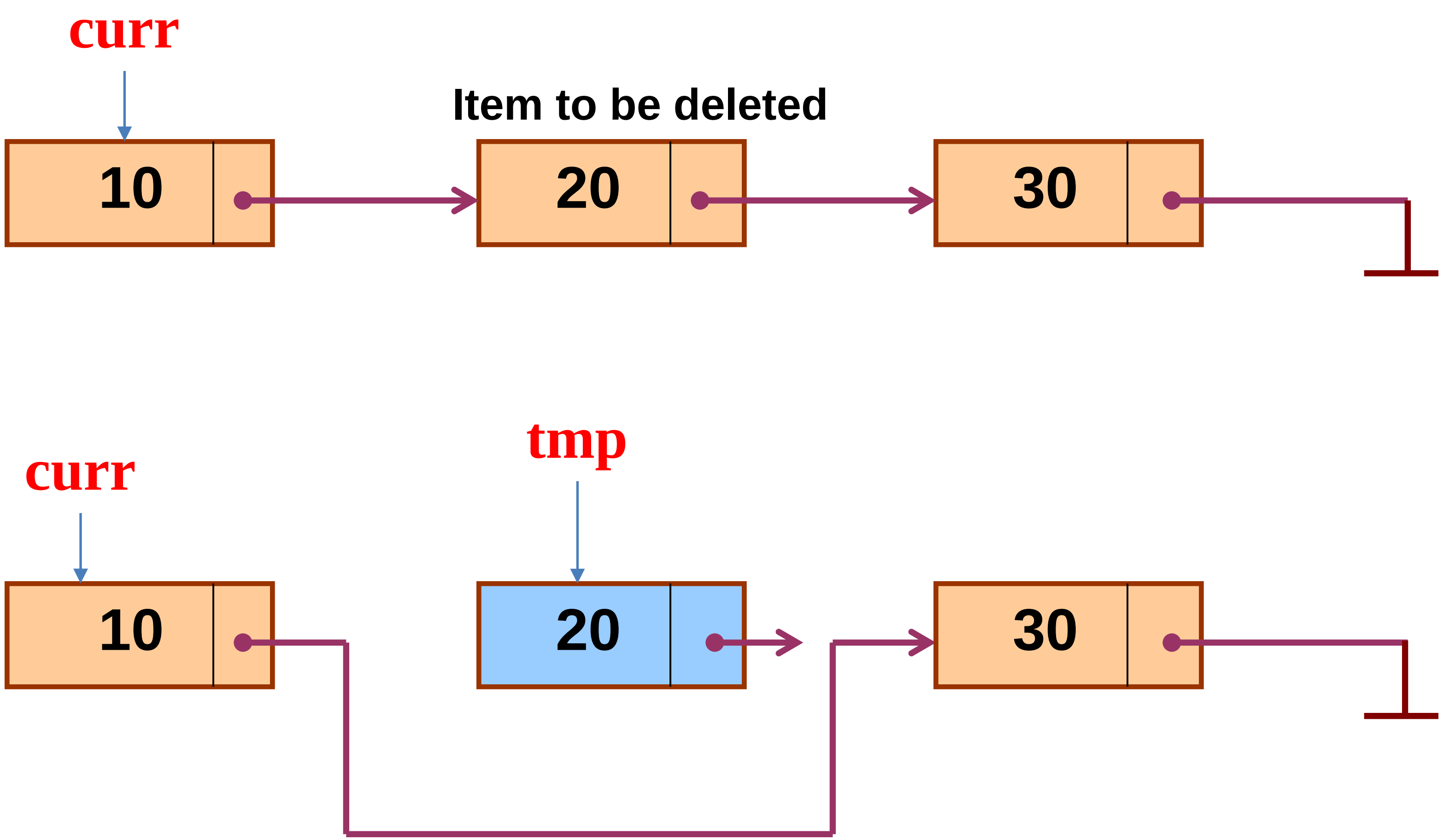


Insert in the middle: code

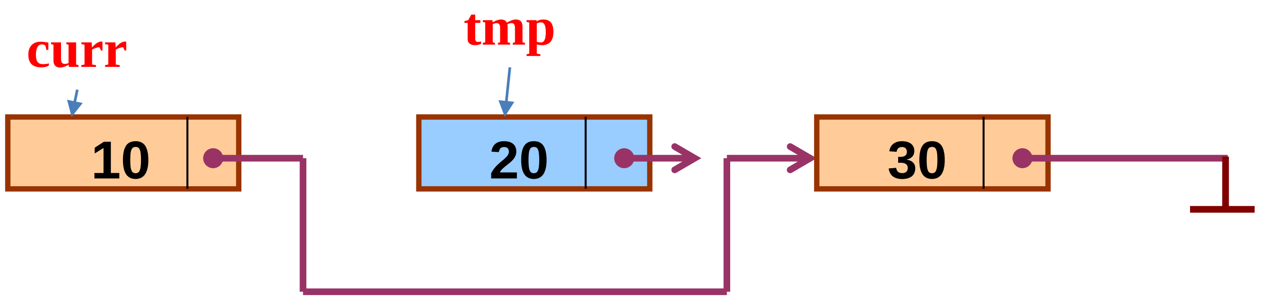



```
typedef struct ListNode {
    int element;
    struct ListNode* next;
} Node;

void insert(Node* curr)
{
    Node * tmp;
    tmp=(node *) malloc(sizeof(ListNode));

    tmp->next=curr->next;
    curr->next=tmp;
}
```

Delete in the middle: Illustration



Delete in the middle: Code




```
typedef struct ListNode {
    int element;
    struct ListNode* next;
} Node;

void delete(Node* curr)
{
    Node* tmp;
    tmp=curr->next;
    curr->next=tmp->next;
    free(tmp);
}
```

In essence ...

- For insertion:
  - A record is created holding the new item.
  - The next pointer of the new record is set to link it to the item which is to follow it in the list.
  - The next pointer of the item which is to precede it must be modified to point to the new item.
- For deletion:
  - The next pointer of the item immediately preceding the one to be deleted is altered, and made to point to the item following the deleted item.

Array versus Linked Lists

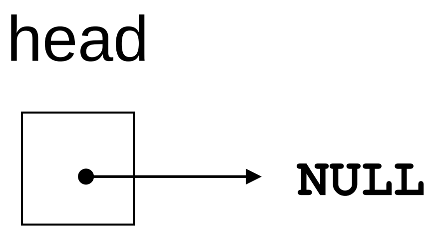
- Arrays are suitable for:
  - Inserting/deleting an element at the end.
  - Randomly accessing any element.
  - Searching the list for a particular value.
- Linked lists are suitable for:
  - Inserting an element.
  - Deleting an element.
  - Applications where sequential access is required.
  - In situations where the number of elements cannot be predicted beforehand.

Linked List Operations

- Basic operations:
  - add a node to the end of the list
  - insert a node within the list
  - traverse the linked list
  - delete a node
  - delete/destroy the list

Empty List

- Empty List
  - A list with no nodes is called the empty list



  - Define a pointer for the head of the list:

```
ListNode *head = NULL;
```

    - Head pointer initialized to **NULL** to indicate an empty list

NULL Pointer

- Is used to indicate end-of-list
- Should always be tested for before using a pointer:

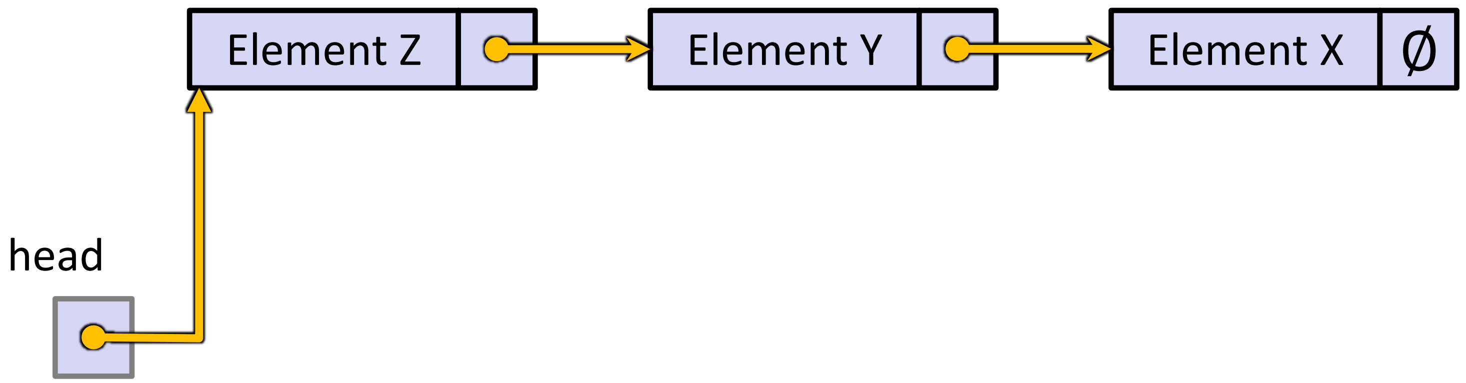
```
ListNode *p;
while (p != NULL)
    ...
```

- Can also test the pointer itself:

```
// same meaning as above
while (!p)
...
```

Simple Linked List in C

- Each node in a linear, singly-linked list contains:
  - Some element as its payload
  - A pointer to the next node in the linked list
    - This pointer is NULL (or some other indicator) in the last node in the list



Traversing a Linked List (1/2)

- List traversals visit each node in a linked list to display contents, validate data, etc.

- Basic process of traversal:

  set a pointer to the head pointer

  while pointer is not ***`NULL`***

      process data

      set pointer to the successor of the current node

  end while

Traversing a Linked List (2/2)

- Pseudo code

  set a pointer to the head pointer

  while pointer is not ***NULL***

      process data

      set pointer to the successor of the current node

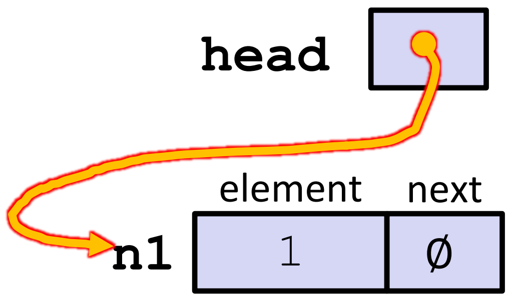
  end while

```
void printList(Node* head) {
    Node* curr = head;
    printf("\[ ");

    while (curr != NULL) {
        printf(" (value: %d) ", curr->element);
        curr = curr->next;
    }
    printf(" ]\n");
}
```

Linked List Node (1/2)

```
#include <stdio.h>
#include <stdlib.h>

typedef struct ListNode{
  int element;
  struct ListNode* next;
} Node;

int main(int argc, char** argv) {
  Node* head = NULL;
  Node *n1, *n2;
  n1=(Node*) malloc(sizeof(Node));
  n1->element =1;
  n1->next = NULL;
  head = n1;
```

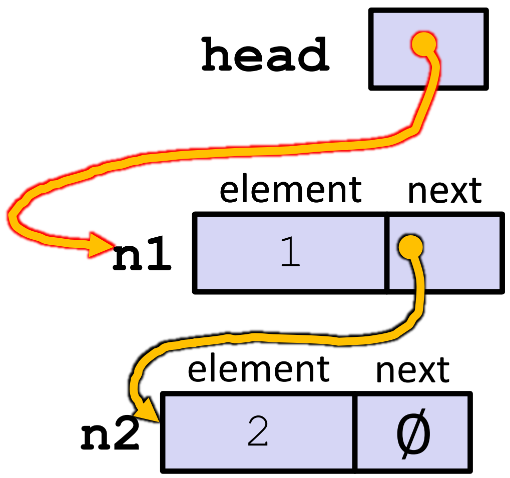
manual_list.c



Linked List Node (2/2)

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct ListNode{
  int element;
  struct ListNode* next;
} Node;

int main(int argc, char** argv) {
  Node* head = NULL;
  Node *n1, *n2;
  n1=(Node*) malloc(sizeof(Node));
  n1->element =1;
  n1->next = NULL;
  head = n1;
  n2=(Node*) malloc(sizeof(Node));
  n2->element =2;
  n2->next = NULL;
  head->next = n2;
  printList(head);
  return 0;
 }
```

manual_list.c



Push Onto List(1/14)

Arrow points to
next instruction.

Push: insert at the head of list

push_list.c

```
typedef struct ListNode{
  int element;
  struct ListNode* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  printList(list);
  return 0;
}
```

(main) `list` Ø

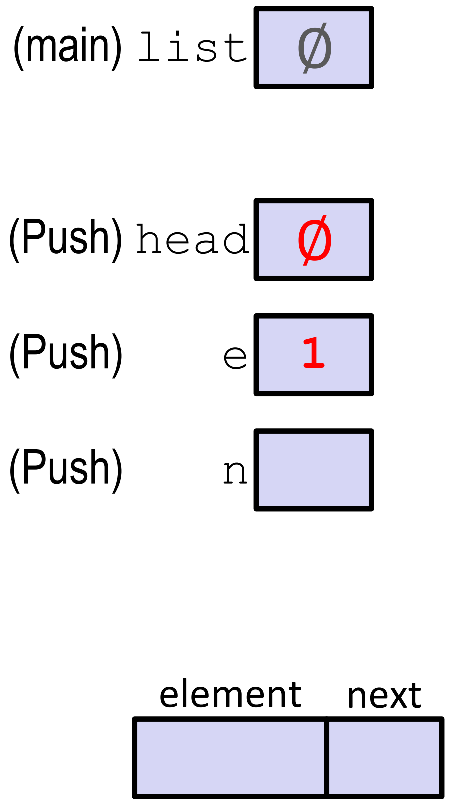
Push Onto List(2/14)

Push: insert at the head of list

Arrow points to *next* instruction.

push_list.c

```
typedef struct ListNode{
  int element;
  struct ListNode* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  printList(list);
  return 0;
}
```



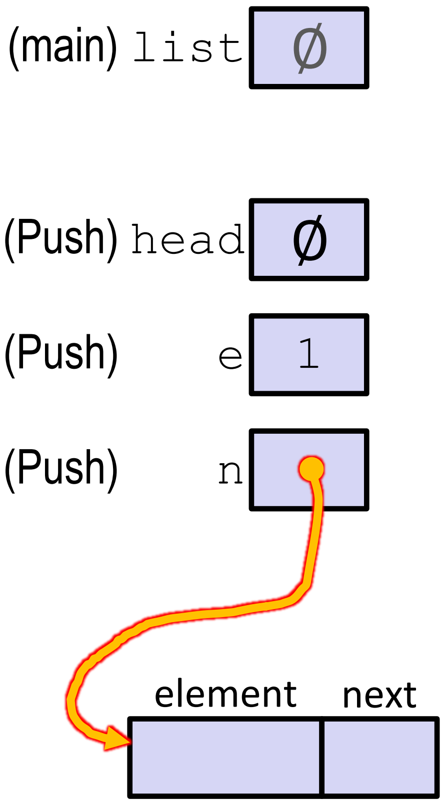
Push Onto List(3/14)

Push: insert at the head of list

Arrow points to *next* instruction.

push_list.c

```
typedef struct ListNode{
  int element;
  struct ListNode* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  printList(list);
  return 0;
}
```



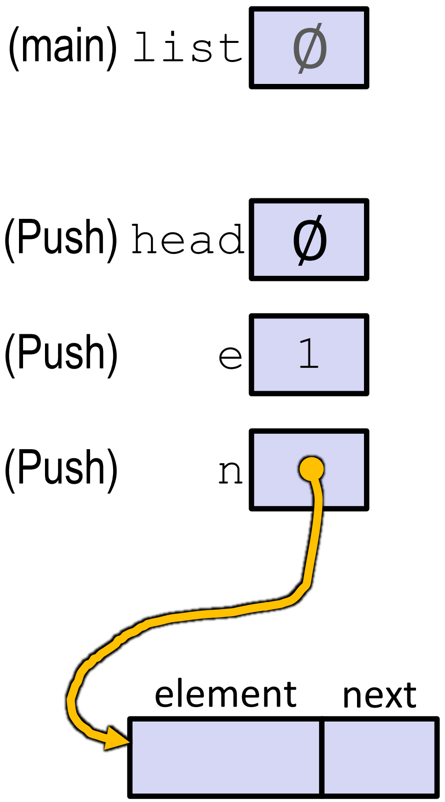
Push Onto List(4/14)

Arrow points to *next* instruction.

Push: insert at the head of list

push_list.c

```
typedef struct ListNode{
  int element;
  struct ListNode* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  printList(list);
  return 0;
}
```



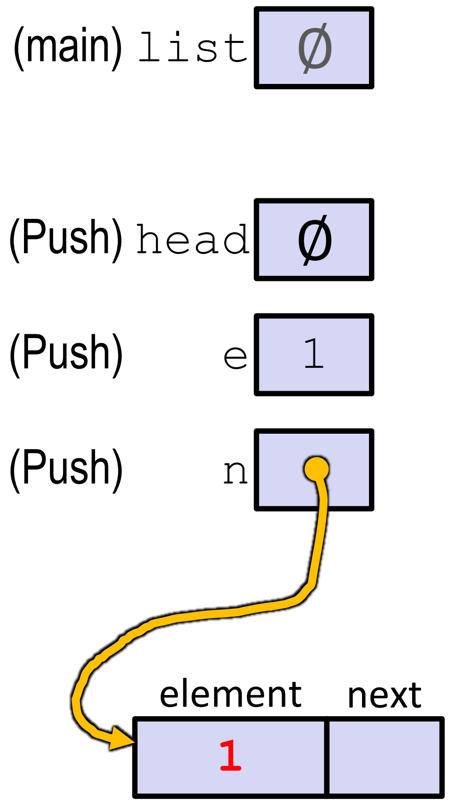
Push Onto List(5/14)

Push: insert at the head of list

Arrow points to *next* instruction.

push_list.c

```
typedef struct ListNode{
  int element;
  struct ListNode* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  printList(list);
  return 0;
}
```



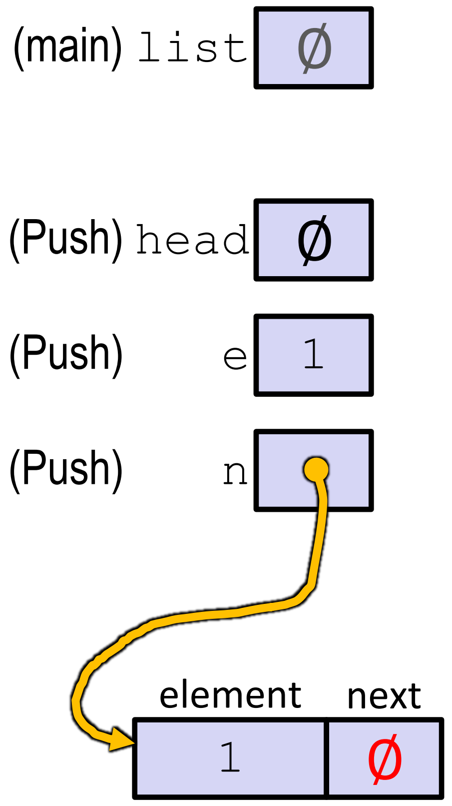
Push Onto List(6/14)

Arrow points to *next* instruction.

Push: insert at the head of list

push_list.c

```
typedef struct ListNode{
  int element;
  struct ListNode* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  printList(list);
  return 0;
}
```



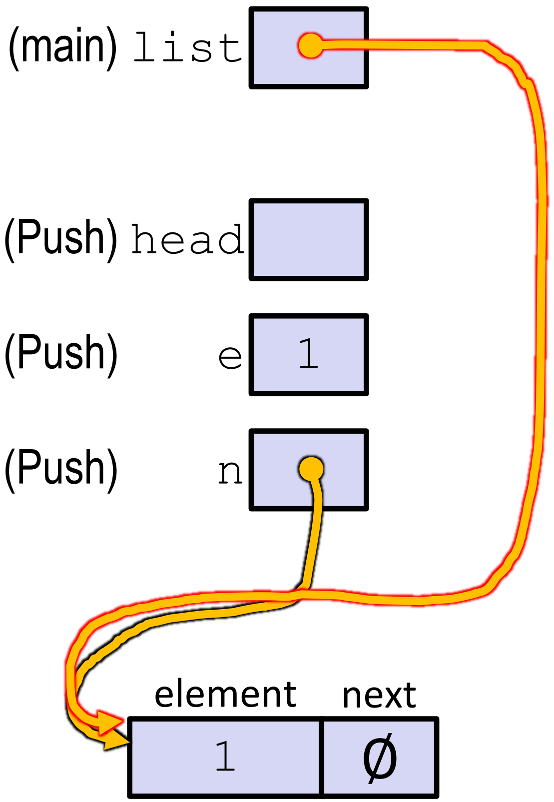
Push Onto List(7/14)

Push: insert at the head of list

Arrow points to
next instruction.

push_list.c

```
typedef struct ListNode{
  int element;
  struct ListNode* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  printList(list);
  return 0;
}
```



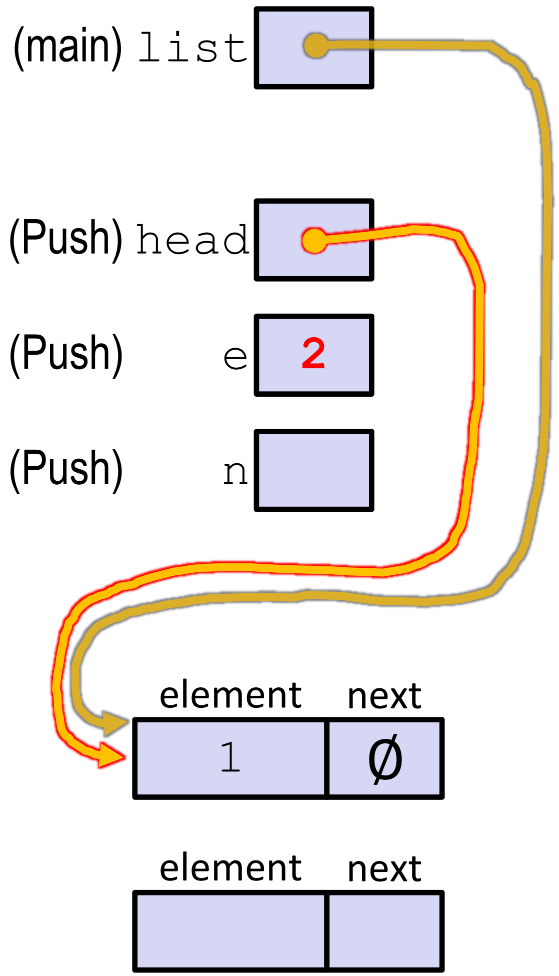
Push Onto List(8/14)

Push: insert at the head of list

Arrow points to *next* instruction.

push_list.c

```
typedef struct ListNode{
  int element;
  struct ListNode* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  printList(list);
  return 0;
}
```



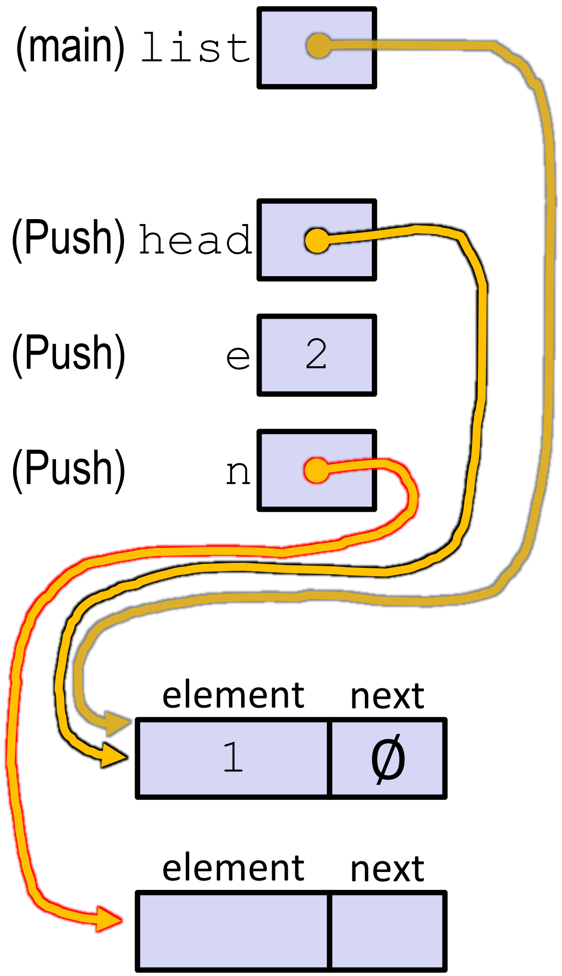
Push Onto List(9/14)

Push: insert at the head of list

Arrow points to *next* instruction.

push_list.c

```
typedef struct ListNode{
  int element;
  struct ListNode* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  printList(head->next);
  return 0;
}
```



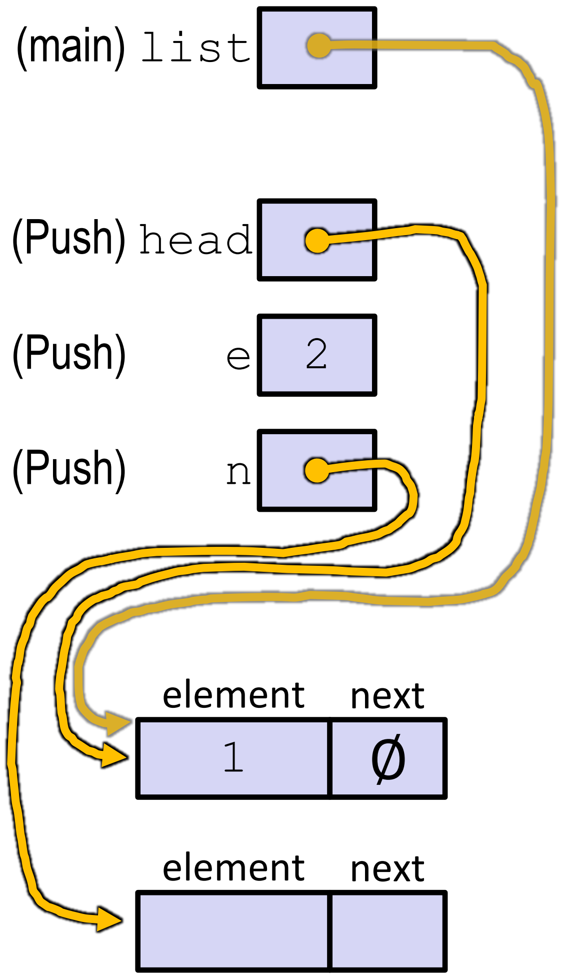
Push Onto List(10/14)

Push: insert at the head of list

Arrow points to *next* instruction.

push_list.c

```
typedef struct ListNode{
  int element;
  struct ListNode* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  printList(list);
  return 0;
}
```



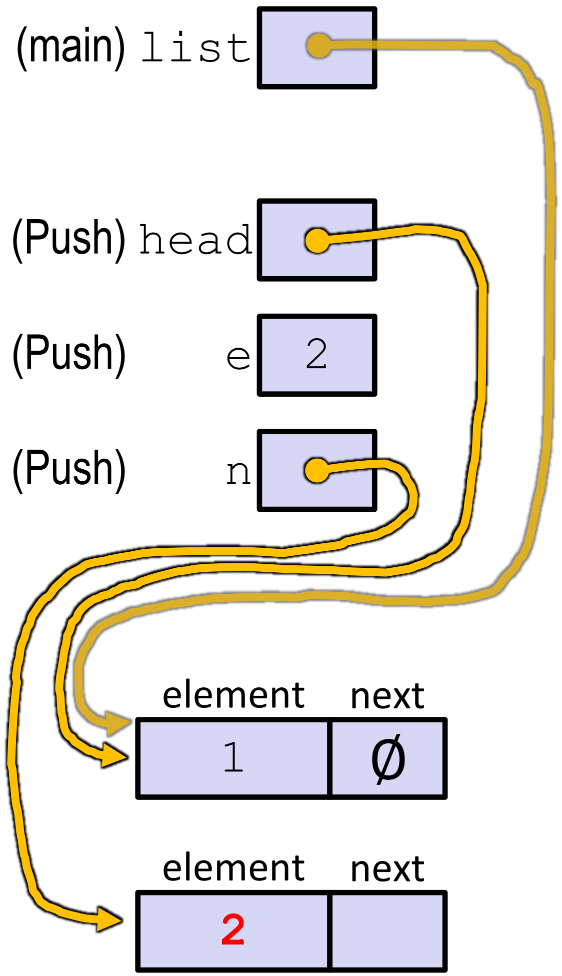
Push Onto List(11/14)

Push: insert at the head of list

Arrow points to *next* instruction.

push_list.c

```
typedef struct ListNode{
  int element;
  struct ListNode* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  printList(list);
  return 0;
}
```



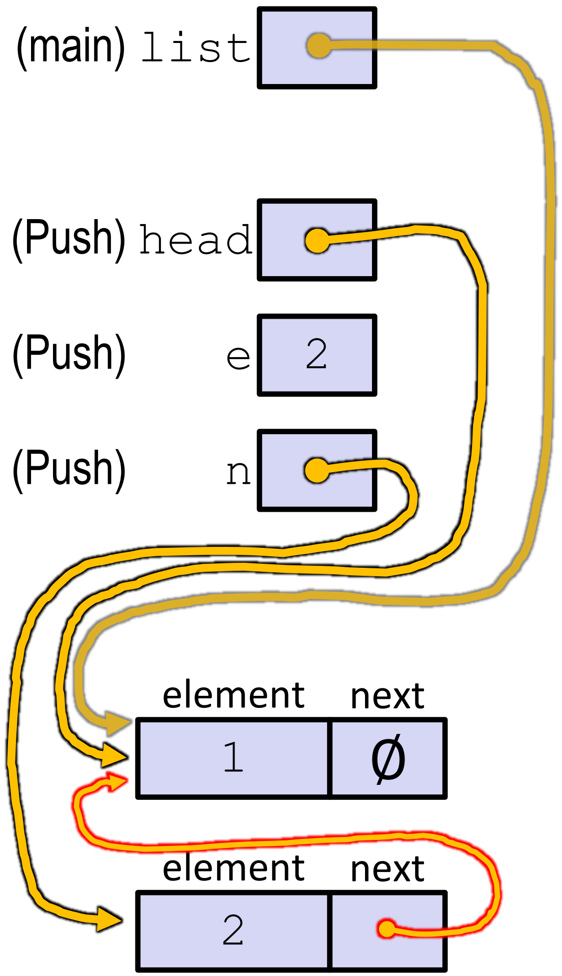
Push Onto List(12/14)

Push: insert at the head of list

Arrow points to *next* instruction.

push_list.c

```c
typedef struct ListNode{
  int element;
  struct ListNode* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  printList(list);
  return 0;
}
```



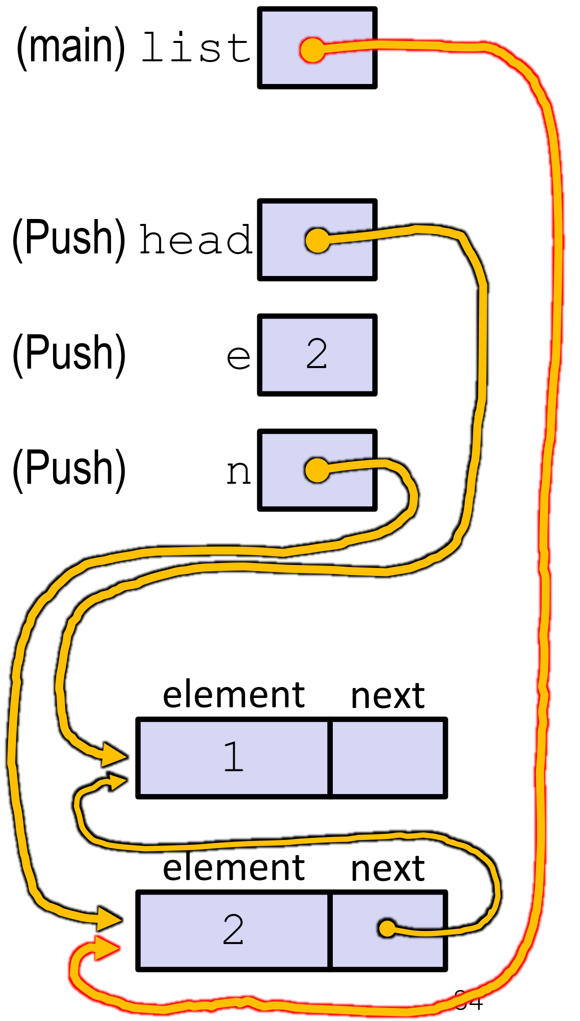
Push Onto List(13/14)

Push: insert at the head of list

Arrow points to *next* instruction.

push_list.c

```
typedef struct ListNode{
  int element;
  struct ListNode* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  printList(list);
  return 0;
}
```



Push Onto List(14/14)

Arrow points to *next* instruction.

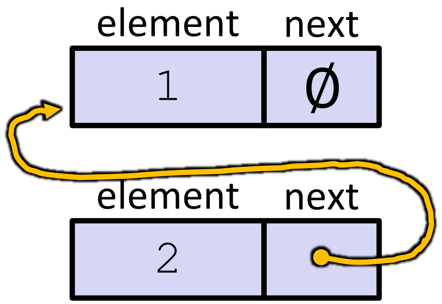
Push: insert at the head of list

push_list.c

```c
typedef struct ListNode{
  int element;
  struct ListNode* next;
} Node;

Node* Push(Node* head, int e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}

int main(int argc, char** argv) {
  Node* list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  printList(list);
  return 0;
}
```

A (benign) memory leak!
Try running with Valgrind:

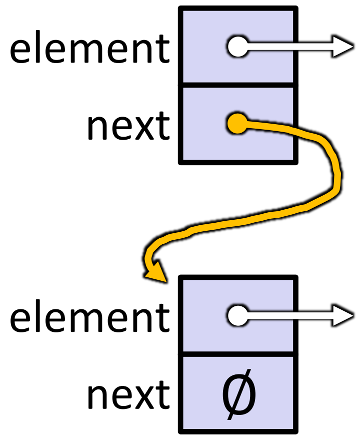
```
bash$ gcc -Wall -g -o
push_list push_list.c

bash$ valgrind --leak-
check=full ./push_list
```



A Generic Linked List

- Let's generalize the linked list element type
  - Let customer decide type (instead of always `int`)
  - Idea: let them use a generic pointer (*i.e.* a `void*`)

```c
typedef struct ListNode {
  void* element;
  struct ListNode* next;
} Node;

Node* Push(Node* head, void* e) {
  Node* n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}
```



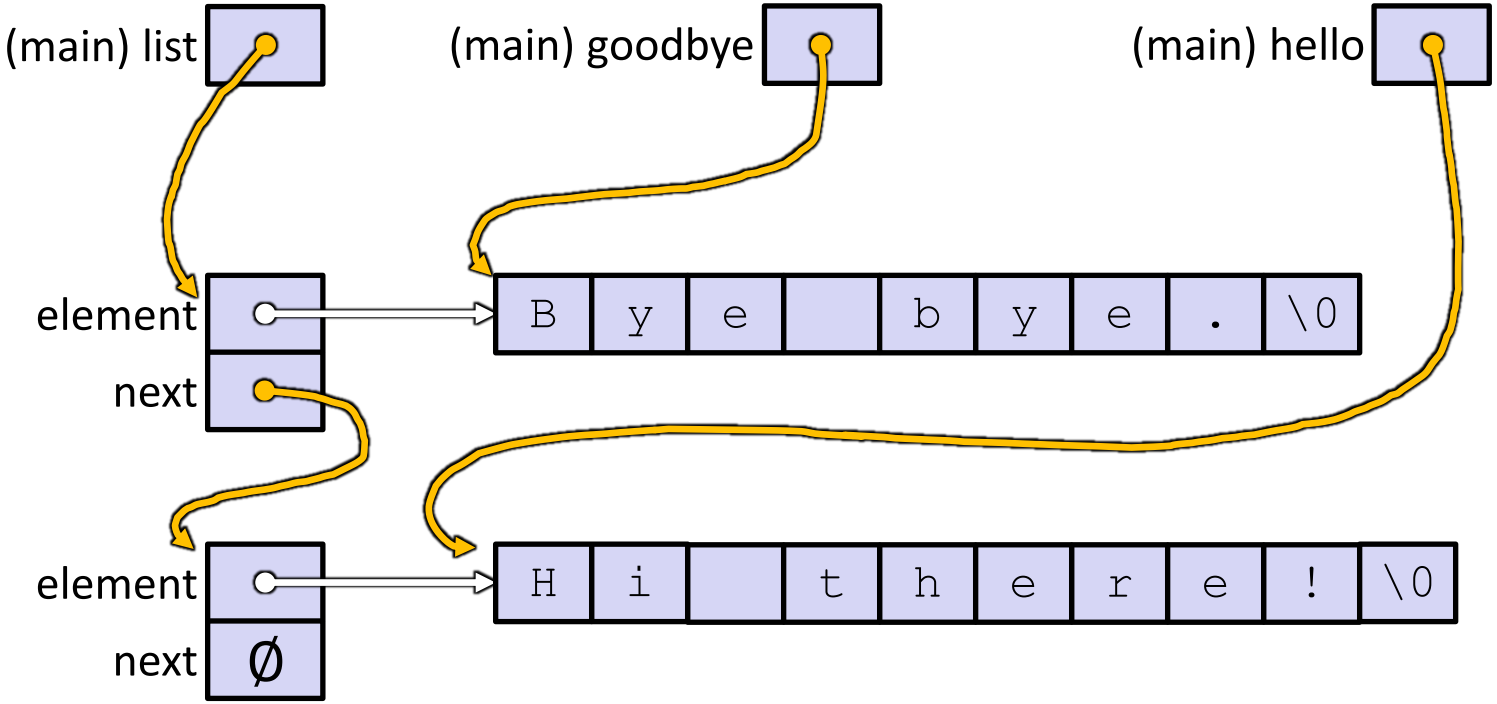
Using a Generic Linked List

- Type casting needed to deal with `void*` (raw address)
  - Before pushing, need to convert to `void*`
  - Convert back to data type when accessing

```c
typedef struct ListNode{
  void* element;
  struct ListNode* next;
} Node;

Node* Push(Node* head, void* e);    // assume last slide's code

int main(int argc, char** argv) {
  char* hello = "Hi there!";
  char* goodbye = "Bye bye.";
  Node* list = NULL;

  list = Push(list, (void*) hello);
  list = Push(list, (void*) goodbye);
  printf("payload: '%s'\n", (char*) ((list->next)->element) );
  return 0;
}
```

manual_list_void.c

Resulting Memory Diagram



Linked List

- Using double pointers
  - https://dev-notes.eu/2018/07/double-pointers-and-linked-list-in-c/
  - https://www.learn-c.org/en/Linked_lists
  - https://www.quora.com/Why-double-pointers-are-used-in-linked-list

- Using Pointer
  - https://dojang.io/mod/page/view.php?id=645 (Korean)

Q&A